

A heuristic approach to query generalisation in semantic databases

Craig I. McGeachie

November 3, 1995

1 Introduction

Imprecise querying is a technique where any database query that has no answer can be generalised into a form that does have an answer. This generalisation is done by carefully relaxing the restrictions specified in the query. Imprecise querying can be performed in a variety of database system types, but the majority of work has been in the relational model.

Semantic databases are an extension of frame-based description languages, and have been the subject of extensive study [2, 3, 14]. Most systems operate on the basis of automatic subsumption where the user specifies concept descriptions, and the database management system then determines the relationship of concepts to each other. The process of subsumption is highly important in the query process as well. If subsumption is intractable, then so is querying. CLASSIC, a system where subsumption is tractable, has been presented in [3].

It has been proposed that a semantic database model is more suited to the area of imprecise querying [1], but this research has paid no attention to the question of computing needs for potentially very large databases. This paper attempts to deal with this by applying ideas of imprecision to CLASSIC. The addition of imprecision to a query system can produce an intractable computational problem, so that any advantages in using CLASSIC are lost. A solution to this, using heuristics, is offered.

In this paper, the following typographic conventions are used. Typewriter font is used to distinguish elements that are part of, or could be part of, the CLASSIC language. Case is used to differentiate between different types

of language components. Uppercase is used for CONSTRUCTORS. Lower case indicates attribute names. Concept names are in mixed case.

The remainder of this paper is organised as follows. Section 2 discusses the reasons for adopting an imprecise querying system. Section 3 presents an overview of related work that has been done with relational databases. Section 4 introduces the idea of semantic databases and query processing in them, using CLASSIC and CANDIDE as examples. Section 5 discusses query generalisation in semantic databases. It presents the application of generalisation rules to CLASSIC. Section 6 presents a technique that can be used to manage computational complexity introduced by imprecise querying. Finally directions for further research are presented in Section 7, and conclusions are reached in the last section.

2 Motivations for imprecise querying

In any situation where learning is done by trial and error, some information as to the trials effectiveness is necessary, and the more the better. If only yes/no information about a trials result is forthcoming, then each trial is essentially random because informed modification is impossible. Random trials are inefficient. As an example, if someone is instructed to guess a number from 1 to n , and is told whether he is too high, too low, or correct then it will take at most $\lceil \log_2 n \rceil$ attempts to find the answer. Random guessing requires $\frac{n}{2}$ attempts on average. If there was no information about a trial result at all, then we could try all possible numbers, and know that we had the right answer at some stage, but not know when.

We face a similar problem when submitting a database query. The knowledge that we have about the database's structure and contents may be insufficient to construct a query that returns useful information. The worst case is the null result: an answer stating that what we have asked for does not exist. This is equivalent to the yes/no answer in the guessing game. In some cases this may be desirable. In banking applications, one of the possible transactions is account debiting. If the specified account does not exist, then we do not wish to debit the next best possibility. This paper is concerned only with cases where imprecision is allowed.

Imprecise querying is where, if there are no answers that exactly fit the query given, the closest approximation is given. When the database management system (DBMS) responds that it can't fill a query, the user can make

the query less restrictive in an effort to retrieve something. The process of relaxing a query's restrictions is referred to as *generalisation*.

The advantage of imprecise querying is that users do not need to maintain a complete mental schema, or knowledge about the context of the database they are working with. They can work with incomplete or inaccurate knowledge, and any query formed will return an answer that provides them with more information concerning the structure of the database. This is a process that can be performed manually by users, by reforming their query in more general terms every time that a null answer is returned. This is, however, very difficult to do properly, and can be very time consuming as many generalisations may have to be made. It would benefit greatly from automation. The DBMS can progressively relax the restrictions of a query until such point that an answer is found. The relaxations required could also form part of the final answer.

As well as all the potential advantages of imprecise querying, there are disadvantages in the form of the extra computational resources needed in terms of both time and space. Generating a generalisation of a query takes additional processing time, and when many generalisations are needed, then the additional time needed may become prohibitive. Each new query generated will require extra space and, like time, this may not be available. This paper will address these issues, and offer solutions.

3 Imprecise querying in relational databases

Amongst modern database systems, the relational model is perhaps the most common. It is therefore reasonable that most research has been with the relational model [10]. Not all the results have been strictly relational. The FRDB system [15] is a good example. It uses the concept of a fuzzy set, which is an extension of fuzzy logic. A fuzzy set A is a set of elements with an associated set membership function $\mu_A()$. The membership function assigns values in the range 0 to 1, to each element of the set. Each query in the FRDB system is a sequence of statements that perform fuzzy retrieval operations on the database. An optional argument of each statement is a threshold value between 0 and 1, which is used to filter the tuples of the result by selecting only those values exceed that of the threshold. The problem with this approach is that it associates scalar values with more abstract values, and that this association is always highly subjective.

An alternative approach to imprecise querying is to formulate a query solely on numerical measures of similarity. Distances are defined among elements of a domain, and by dividing each distance by the diameter of the domain, a value from 0 to 1 is found that measures *dissimilarity*. This technique is used by ARES [5] and VAGUE [9]. The problem with this approach is again that of assigning quantitative values to qualitative or symbolic data. Again, these associations are highly subjective.

SEAVE [8] is a system that provides an imprecise querying facility for standard (precise) relational database systems. A component of SEAVE is a *supposition generaliser* that generalises queries by weakening mathematical conditions or removing non-mathematical ones altogether. It is assumed that three values are associated with each numeric attribute. These values are its upper bound, lower bound, and an arbitrary step value to specify the minimal weakening of a mathematical condition. For example, the attribute *SALARY* may have minimum 10000, maximum 50000, and step 1000. Therefore a constraint such as *SALARY* < 40000 is generalised to *SALARY* < 41000. If an attribute is non-numeric, it is generalised by dropping the constraint. Imprecise querying is achieved by continually generalising the constraints in the query until a match is found.

The examples above have highlighted a problem with imprecise querying in the relational database schema. Non-numeric values are dealt with badly. At best they are dealt with in an arbitrary and subjective manner. At worst, they are ignored totally. Even the choice of step size that is required by SEAVE for numerical attributes, is arbitrary. The relational format does not provide an explicit representation of the semantics of the data in the database. There are no explicit notational devices that make the task of generalisation, which is concerned with the semantics of the data, easier.

4 Semantic databases

Semantic databases consists of two types of objects: instances, and classes. Information about a class includes information about its parent(s), or most specific predecessor(s). Also included is information about attributes, and their restrictions. These provide necessary, and possibly sufficient conditions, for an instance to be a member of this class. An instance has attributes, which may have values specified for them. Many such databases are based on the KL-ONE system, and form the KL-ONE family [14]. Two

systems that are of especial interest in this paper are CANDIDE [2] and CLASSIC [3].

Semantic databases can be related to more traditional relational types. A concept can be thought of as a table, and contains roles instead of attributes. Instances are used in place of tuples. This provides a rough analogy to help understand why semantic models are better for query generalisation. The key difference is in the way that tables or concepts are related to each other. In a relational database, a table has an attribute that contains a foreign key, such as an employee ID for a supervisor in a record about a department of an organisation. The semantic model also allows this in the form of an instance of the concept of employee to fill the supervisor role of an instance from the department concept.

The semantic model allows concepts to be related in another way; a concept can be a descendant of another concept and inherit structure from it. As an example, the concept of person could be defined as in Figure 1. This can be read as saying “a person has a name, an age, and a height,” that these are the defining characteristics of this concept PERSON. This definition says nothing about what to actually do with the roles of name, age, and height, but it can be safely assumed that they correspond to a text string and two numbers.

```
Person:
    name
    age
    height
```

Figure 1: Definition of PERSON

Now suppose we have a need for the concept of a student. This can be defined as in Figure 2. The difference is in the first line, where the definition of Student references that of Person. This can be read as “a student looks just like person, which we defined elsewhere, but has the additional attributes of a major, and a grade point average.”

What we have done is to create the concept of Person, and then we identified the concept of Student as being a subconcept. This means that every instance of a student is automatically an instance of a person. This is

```

Student: Person
        major
        grade-point-average

```

Figure 2: Definition of STUDENT

information that cannot easily be captured in a relational format; there is no way to say that every student is automatically a person.

Semantic data models can therefore use generalisation techniques that can be applied to relational databases. There are additional techniques available that relate to the hierarchical structure, so semantic databases have a greater choice of generalisations that can be applied to a query, and the advantages of this will be explained in Section 5. The purpose of the following sections is to introduce the idea of semantic databases, using CANDIDE and CLASSIC as examples.

4.1 Query by subsumption

When information is extracted from a semantic database, the process of subsumption is used. A class \mathcal{F} subsumes class \mathcal{G} if and only if every instance of \mathcal{G} is also an instance of \mathcal{F} .

$$\begin{aligned} \forall g \in \mathcal{G} : g \in \mathcal{F} \\ \Rightarrow \mathcal{F} \triangleright \mathcal{G} \end{aligned}$$

which is read as \mathcal{F} subsumes \mathcal{G} . An instance g is an instance of \mathcal{G} if its attributes all satisfy the restrictions specified for \mathcal{G} . We can allow a small misuse of language, and say that \mathcal{G} subsumes g . The subsumption relationships of every instance and class in a database forms an acyclic directed graph called the database taxonomy.

The process of constructing a taxonomy can be automated. The user need not explicitly specify which class subsumes which, but the DBMS can deduce these relationships from class and instance definitions. As a result the taxonomy no longer needs to be under the direct control of the user, and can be maintained in the background. New classes may be defined and inserted on the fly.

To understand why this is useful in the area of query generalisation, we now examine how a query is performed in a semantic database of the KL-ONE family [14]. The answer required for a database query is any instance, or tuple, that fits the restrictions specified by the query. The query is treated as a concept¹, and fitted into the concept taxonomy. For example, Figure 3 shows how we can specify that we want the DBMS to show all the instances that are students over the age of 65.

```
?: STUDENT
   Age (> 65)
```

Figure 3: A query to find students older than 65

Because the DBMS can automatically insert this anonymous class into the taxonomy, it can then trace the subsumption relationships to discover what instances are members of this class. These instances are the answers to the query.

4.2 CANDIDE and CLASSIC

To further explore the concept of the semantic database, it will be helpful to look at two concrete examples, CLASSIC and CANDIDE. These two systems have been chosen because the issue of imprecise querying has been explored in CANDIDE [1], and CLASSIC has been written with a the need for a tractable subsumption process in mind [3]. These two issues will be dealt with later.

A BNF describing the syntax of CANDIDE is presented in [2], but a brief overview will be presented here.

Figure 4 presents an example of a class definition in CANDIDE. The example presents only a limited number of features of the grammar. The definition shown describes the necessary and sufficient conditions to be an engineering graduate student. Engineering graduates are a subclass of the

¹In some systems, the data definition language and the data manipulation language are the same. This means that queries really do look like concepts, but even if the two languages are different, this is no reason for concern

```

ENG-GRAD CLASS
  SUPERCLASS: GRAD
  ATTRIBUTE RESTRICTIONS
    COURSE-LOAD: SOME 3
      DOMAIN CLASS 'ENGINEERING-COURSE'
    ADVISOR: SOME 1
      DOMAIN CLASS 'PROFESSOR'

```

Figure 4: An example of a class definition in CANDIDE

class of graduate students, must be enrolled in least 3 engineering courses, and have at least 1 professor advising them. The superclass field provides for explicit definition of the taxonomy, as would the subclass field. Both of these fields are optional. The superclass field can be used to determine exactly which attributes must be inherited. The attribute restrictions fields, which are also optional, can be used to determine superclass and subclass relationships. Thus CANDIDE provides for redundancy based on a class' inheritance and its structure.

It should be noted that attribute restrictions all have a quantifier, such as *SOME*, *AT-MOST* or *ALL*, that place a cardinality constraint on the values that an attribute can have. For example, an engineering graduate must be doing at least 3 courses. The other restriction placed by every attribute constraint is on the value set. A domain is given, from which every value for an attribute must be drawn. For example, all engineering graduates must have some engineering courses.

We now turn to a short introduction to CLASSIC, which is fully described in [3].

Figure 5 presents an equivalent definition to that presented in Figure 4, but defined using the grammar of CLASSIC. The first thing of note is the separation of the data manipulation and data definition languages. This has little or no relevance to the problem of query generalisation. Generalisation is concerned with modifications of a concept described using the data definition language. It does not matter if there is a separate language that is defines data manipulation, as in the case of CLASSIC. The CLASSIC syntax for describing concepts is heavily influenced by LISP, and, as one might imagine,


```

define-concept[ENG-GRAD,
              (AND
               GRAD
               (AT-LEAST 3 course-load)
               (ALL course-load ENGINEERING-COURSE)
               (AT-LEAST 1 advisor)
               (ALL advisor PROFESSOR))];

```

Figure 5: An example of a class definition in CLASSIC

allows for good use of recursion. One obvious difference to CANDIDE is the presence of an explicit conjunctive. The conjunctive for CANDIDE's attribute restrictions is implicit. No provision is made for explicit definition of superclasses or subclasses. Instead, the inclusion of the named concept GRAD means that for an instance to fulfil the requirements of membership in this class, it must also fulfil all the requirements for the class GRAD. The DBMS can then conclude that the new class inherits all the structure of the superclass, and is a direct subclass of that superclass.

So far it would seem that CLASSIC has, at least, the expressive power of CANDIDE. This is not the case. In CANDIDE, any quantifier could have a domain associated with it. In CLASSIC, attribute restrictions are separated with respect to cardinality and value set constraints. The AT-LEAST constructor can only specify a quantity, not a value set, and the opposite is true of the ALL constructor. This means it is impossible to define a concept Fruitbowl which is required to have at least three apples and two bananas, as well as any other form of fruit that may be there. The best that can be done is that all the fruit be apples or bananas, and there be five pieces. This will limit the ability to generalise a CLASSIC query, compared to CANDIDE, but there are other options that remain open.

A feature that CLASSIC has, that isn't found in CANDIDE, is the ONE-OF constructor. It allows the definition of a concept that is simply a grouping of instances. An example is shown in Figure 6.

```
define-concept[CAR,
               (ONE-OF Ford Volvo Toyota)]
```

Figure 6: An example of a class definition using ONE-OF

4.3 Issues of tractability

For a database to be of practical use, it needs to be able to process a reasonable quantity of data in a reasonable time. Thus the issue of tractability arises, with respect to semantic databases. In systems like CLASSIC and CANDIDE, subsumption is the major operation. There has been extensive exploration of the computational behaviour of such systems [4, 6, 12], with the conclusions are that the complexity of algorithms for subsumption and query processing are very sensitive to choice of concept constructors.

CANDIDE is essentially an extension of the \mathcal{FL} -, KANDOR, and BACK frame-based knowledge representation languages. It has been shown that KANDOR, and consequently CANDIDE, has a trap where complexity gets out of hand [11]. CLASSIC was written with the issue of complexity in mind. This is why it doesn't include expensive features such as OR and NOT, and lacks some of the expressiveness of CANDIDE.

It should be noted that the complexity of subsumption is measured with respect to the size of the concept descriptions, rather than the size of the database. The database size is not an issue in systems such as CLASSIC, which perform a great deal of preprocessing, and maintain the subsumption hierarchy, so that once a concept is fitted into the taxonomy, it is already linked to all the instances and classes that it subsumes.

Because of the complexity problem with subsumption in CANDIDE, it isn't suitable for large, complicated situations, while at the moment it would seem that CLASSIC is. It would also seem desirable to merge the ideas of tractable query subsumption and imprecise querying, so the next step is to explore the potential for query generalisation in CLASSIC.

5 Generalisation of semantic databases

Generalisation is the process of modifying the intension of a concept, so that the new concepts extension is potentially larger than the original, and completely includes everything specified by the original concept. When a database query is generalised, care must be taken to avoid over generalisation which leads to the problem of the *family resemblance effect* [13]. As a class becomes more inclusive, its instances will have less in common with each other, and there will be less knowledge about the database that can be inferred from them. If the original query produces a null answer, then it is preferable to return some small portion that is close rather than the entire database. In other words, the best alternative to the null answer is the one yielded by smallest degree of generalisation, and this should provide the user with the best insight into the databases structure. Consequently the larger the variety of generalisations available at any given stage, the better. The more ways of producing alternative descriptions, the greater the likelihood that one of them will produce an answer, so fewer levels of generalisation will be needed.

A good description of rules for generalisation may be found in [7]. Many ideas on concept generalisation from Section 5.2 of this paper will be repeated here in Sections 5.1.1 to 5.1.5 where their applications in CLASSIC are presented. As well as referring to the more general generalisation rules, comparisons will be drawn with CANDIDE, wherever valid.

5.1 Generalisation in CLASSIC

This section will describe the practical application of generalisation rules to CLASSIC queries. The form of CLASSIC lends itself to some generalisation rules, but not others. For example, the lack of an OR constructor makes it impossible to directly transform a conjunction into a disjunction. The use of a ONE-OF constructor makes it possible to apply the extending reference rule, which can't be done with CANDIDE.

CLASSIC does not have an EXACTLY constructor which is a quantity restriction that requires a field to have exactly a specified number of values. This lack can be dealt with by using a combination of AND, AT-LEAST, and AT-MOST. When the quantification domain of either AT-LEAST or AT-MOST is extended, or either the AT-LEAST or AT-MOST condition is dropped, then the effect is that of the artificially created exactly operator being generalised

with the extension of quantification domain rule. This is called an *emergent generalisation*.

Before a description of the more specific generalisation rules, a discussion of their application in light of the recursive nature of the definition of CLASSIC is appropriate. The AND, ALL, PRIMITIVE, and DISJOINT-PRIMITIVE constructors all take arbitrary concept expressions, which may consist of further recursive constructs. As mentioned earlier, overgeneralisation is to be avoided, therefore these constructors must be generalised at the lowest possible level. If a concept expression to be generalised contains some other concept expression, then the outer expression is generalised by generalising the inner expression. Consider the following:

$$\begin{array}{ccc} (\text{AND} & & (\text{AND} \\ (\text{AT-LEAST } 2 \text{ courses})) & \parallel < & (\text{AT-LEAST } 1 \text{ courses})) \end{array}$$

where $\parallel <$ denotes a generalisation. The AND constructor contains a concept expression, that can itself be generalised as discussed in Section 5.1.1. The AT-LEAST term cannot now be generalised further, so generalisation of the entire expression proceeds at the outer level by dropping conditions as discussed in Section 5.1.3. This idea is easily extended to more levels of nesting.

5.1.1 Extending the quantification domain

A cardinality constraint is generalised by applying the ‘extension of quantification domain’ rule discussed in [7]. Michaliski introduced the concept of a numerical quantifier, expressed in the form

$$\exists(I)v, S[v]$$

where I , the index set, denotes a set of integers and $S[v]$ is an expression having v as a free variable. The previous expression evaluates to true if the number of values of v for which the expression $S[v]$ is true is an element of the set I . Consequently (AT-LEAST 3 courses) can be rewritten as

$$\exists(\geq 3)v, \{v | \text{attr-value}(\text{courses}, v)\}$$

which is read as there exist 3 or more values for v such that v is an attribute value of **courses**.

Using the concept of the numerical quantifier, the extension of the quantification domain rule can be expressed as

$$\exists(I_1)v, S[v] \parallel \exists(I_2)v, S[v], I_1 \subseteq I_2$$

We can now define how the two types of cardinality constraints in CLASSIC are generalised.

- (AT-LEAST n $\langle \text{role-expr} \rangle$) \parallel (AT-LEAST $(n - 1)$ $\langle \text{role-expr} \rangle$), $n > 1$
We can generalise the requirement that we have n or more attribute values for a role to the requirement that we have $n - 1$ or more values. This is also expressed as

$$\exists(\geq n)v, S[v] \parallel \exists(\geq (n - 1))v, S[v]$$

It should be noted that n must be greater than 1. If n is equal to 1, then this expression cannot be generalised further, and generalisation must continue with the next outermost nesting level.

- (AT-MOST n $\langle \text{role-expr} \rangle$) \parallel (AT-MOST $(n + 1)$ $\langle \text{role-expr} \rangle$)
This is identical to the case for AT-LEAST except that the upper-most end of a range of integers is being extended.

$$\exists(\leq n)v, S[v] \parallel \exists(\leq (n + 1))v, S[v]$$

Note that, unlike the case for AT-LEAST, there is no specified limit to the extension of integer sets because of the open world assumption in CLASSIC. This will cause a problem, in that there is no point at which this term can no longer be generalised, and thus no point at which generalisation of higher level concept expressions can take place. In fact, it is possible to construct a query for a database which can never be satisfied, no matter how far it is generalised, and generalisation will never stop. This problem can be solved by allowing a special case, where both the quantification domain for AT-MOST is extended, and appropriate generalisation rules are applied to the next outermost nesting level under the assumption that this AT-MOST concept expression cannot be generalised further.

In comparison to CANDIDE, the ALL concept constructor cannot be generalised by extending the quantification domain. In CANDIDE, attribute

constraints are built from a value constraint and a quantity constraint, the combination of which is considered atomic. This allows the specification of the existential quantifier by writing `SOME 1 <v>`, where v is some value constraint. It is read to mean that an attribute has at least 1 value of type v , and may have other values. There is no such construct in CLASSIC.

5.1.2 Climbing the generalisation tree

A concept expression can be as simple as a named concept. Any named concept will already have been processed and placed into the database taxonomy by the CLASSIC classifier. As such, its parent concepts are known. If \mathcal{F} subsumes \mathcal{G} then all instances that are subsumed by \mathcal{G} are also subsumed by \mathcal{F} . A concept consisting of a named concept can be generalised by replacing it with the name of the parent concept. We can write this as

$$\langle \text{concept} \rangle \parallel < \text{parent}(\langle \text{concept} \rangle)$$

As an example, if `Car` is subsumed by `Vehicle` then

$$\text{Car} \parallel < \text{Vehicle}$$

is a valid generalisation. If we also use the rules concerning the nesting of concept expressions, then we can also perform the generalisation

$$(\text{ALL thing-driven Car}) \parallel < (\text{ALL thing-driven Vehicle})$$

In CANDIDE, climbing the generalisation tree is used to relax the value set constraints of attribute restrictions. A query class also specifies a superclass. This superclass specification can also be generalised by climbing the generalisation tree.

5.1.3 Dropping condition rule

CLASSIC provides the `AND` constructor for composition, which will accept one or more concept expressions as arguments. For any instance to satisfy a concept constructed with `AND`, it must satisfy all the sub-concepts. Any such concept can be generalised by simply dropping one of the sub-concepts, which have been conjunctively linked.

$$(\text{AND } \langle \text{concept} \rangle^n \langle \text{concept} \rangle) \parallel < (\text{AND } \langle \text{concept} \rangle^n), n \geq 1$$

It is not explicitly stated in [1] that the dropping condition rule is used in CANDIDE, but when an attribute restriction cannot be generalised any further, then the attribute restriction is dropped altogether.

5.1.4 Extending reference rule

CLASSIC does not provide an OR constructor. This means that external disjunction is impossible. Internal disjunctions are possible however. The ONE-OF constructor takes a number of instances as arguments, and defines a class that is a time invariant set of the objects specified.

A concept, that has been defined by using ONE-OF, fits into the taxonomy at some point. The concept can be extended, or generalised, by adding a new instance to the ONE-OF construction. The general extension of reference rule is expressed as

$$\text{CTX} \wedge [L = R_1] \parallel < \text{CTX} \wedge [L = R_2], R_1 \subseteq R_2 \subseteq \text{domain}(L)$$

where CTX stands for some arbitrary concept expression, L is a term, and R_1 and R_2 (references) are internal disjunctions of values of L . References R_1 and R_2 can be interpreted as sets of values that descriptor L can take in order to satisfy the concept description. The rule states that a concept description can be generalised by enlarging the reference of a descriptor ($R_2 \supseteq R_1$). The elements added to R_2 must, however, be from the domain of L .

From this definition, we can see that the generalisation

$$(\text{ONE-OF } I) \parallel < (\text{ONE-OF } I \ i), \mathcal{F} \triangleright I, \mathcal{F} \triangleright i, i \notin I$$

where I is the original set of instances, \mathcal{F} is the most specific named concept that subsumes all the instances in I , and i is some instance that \mathcal{F} subsumes and is not included in I , can be applied in CLASSIC. If there are no more values i that satisfy the stated restrictions, then ONE-OF cannot be generalised any further.

CANDIDE has no constructor that equates to ONE-OF, so no comparisons can be drawn.

5.1.5 Turning conjunction into disjunction rule

Because of the lack of an OR constructor in CLASSIC, this form of generalisation is not directly possible, but the effects can be achieved. In general,

this rule can be stated as

$$F_1 \wedge F_2 \parallel < F_1 \vee F_2$$

where F_1 and F_2 are arbitrary descriptions.

A query generalisation can produce several new queries. When the dropping condition rule is applied, any conjunctive term can be dropped. If two new queries are produced by dropping different terms, then their extensional answers can be combined into one answer. This answer is the same that would be produced by turning the conjunction into a disjunction.

The choice of which term to drop is arbitrary, and if the query generalisation system simply returns the first answer produced by dropping a term without checking the results of dropping other terms, then any answer produced by dropping other terms will not be returned. These alternative answers may be desired, so it is possible that there may be occasions when a disjunctive answer combining two or more answers produced by the dropping of terms is preferable to an individual answer set.

As stated, CLASSIC does not provide an OR constructor. Although it is possible to work around this as described, this is contrary to the design of CLASSIC. Any answer produced by using a disjunction cannot be described as a concept expression in CLASSIC notation, so this rule should only be used in exceptional circumstances.

5.2 Issues of tractability

In Section 4.3 it was explained that in order for a database management system to be useful, it needed to be able to process queries in a reasonable amount of time. It also needs to be able to do this with large amounts of data. This is the reason cited for choosing CLASSIC over CANDIDE as a base DBMS. The advantages of CLASSIC are wasted if introducing the technique of query generalisation is going to make query processing intractable again. Unfortunately this may happen. Consider the following informal argument.

It has been shown that when a query is processed in an ordinary fashion, that is, no generalisation is performed, that the main factor affecting processing complexity is the subsumption process. When queries are generalised, then any null answer results in one or more new queries being produced, which are then subjected to the process of subsumption. If these new queries all produce null answers, then the cycle repeats until such time as a non-null

answer is produced. If k new queries are produced, on average, from each query that yields a null answer, then after the first round of generalisations, there will be k new queries to be processed. Assuming these all produce null results, then the next round, there will be k^2 new queries. If queries have to be subjected to n levels of generalisation, on average, then there will be k^n queries that must be fitted into the database; query generalisation is subject to exponential growth. At this point it seems reasonable to assume that as queries become more complex, the number of levels of generalisation required, on average, will increase. To put this another way, n is proportional to the complexity of concept expressions, measured with respect to the number of terms.

6 Heuristics for choosing generalisations

The combinatorial explosion problem renders the straight forward approach to generalisation presented in [7] impractical for a general use database. In order to render query generalisation a useful technique, its complexity must be managed in some way. This section presents heuristics and an algorithm to achieve this.

The first step is to assign an order of preference to each possible type of generalisation rule that may be applied to a query. The choice of this ordering is somewhat arbitrary, and may differ depending on individual requirements. Presented here is an ordering chosen on the basis of subjective opinion as to which yields the least relaxation.

1. Extend the quantification domain of an AT-LEAST constructor.
2. Extend the reference of a ONE-OF constructor.
3. Replace a named concept with the name of its parent.
4. Drop a condition.
5. Extend the quantification domain of an AT-MOST constructor.

Extending the quantification domain of the AT-LEAST constructor is the most preferable, compared to extending the quantification domain of the AT-MOST constructor, which is least preferable. This is because there is no upper bound for the extension to AT-MOST, and the solution is to both extend the

quantification domain, and generalise the concept expression at the next outermost nesting level. This can potentially achieve a much greater degree of relaxation than any other rule. The rule of turning a conjunction into a disjunction is not included here, as it is impossible to express a disjunctive concept in CLASSIC, and thus it is impossible to produce a query that can be generalised further. This does not preclude the returning of a disjunctive answer in certain situations as outlined in Section 5.1.5. Any of these rules may be applied to a query in more than one way. When this is the case, then the choice between them is arbitrary, as it is assumed that the generaliser knows nothing about the relative importance of different attributes.

For any query, there is a maximum number of new queries, that can be produced by generalisation, that is directly proportional to the complexity of the original query. As a query is repeatedly generalised, its complexity is non-increasing. Let q be some query. Let \hat{q} be the set of queries that can be produced by applying the rules described in Section 5 to q . The rules used to produce \hat{q} have a precedence ordering as described above. If all the queries in \hat{q} have null answers, then we can use this ordering to select one member from \hat{q} , and label it q' . The set of generalisation rules can now be applied to q' to yield \hat{q}' . This process can be repeated until a query with a non-null answer is produced. Limiting the number of new queries generated at each step by choosing one query to generalise, means that the number of queries generated is bounded by a constant based on the complexity of the original query.

It is worth noting an interesting effect related to the idea of family resemblance that will be used as a justification for heuristics. At each level of generalisation, the number of queries grows exponentially, but consider the following two CLASSIC concept expressions.

(AT-LEAST 2 Thing)

(AT-LEAST 1 Course)

If the taxonomy specifies that **Thing** is the direct superclass of **Course**, then both of these concept expressions will generalise to

(AT-LEAST 1 Thing)

Thus, although the query generalisation tree grows in width exponentially with respect to its depth, eventually many of the new queries are identical to each other, and if only unique queries are included at each level, then the tree becomes a lattice that initially expands as it grows downward, and then begins to contract. In fact, eventually every concept expression will eventually generalise to

Thing

or its semantic equivalent, so that the query generalisation graph narrows to single points at the top and bottom.

Because the generalisation tree eventually converges despite the initial divergence, it would seem reasonable to allow heuristics involving the selection of only one generalisation at each level. A caveat to be observed is that the convergence may not provide beneficial effects for computational tractability. Even though two queries produced by generalisation may be the same, recognising that they are identical requires effort, which may require more computing power than simply generalising from both points and ignoring any possibility of redundancy. This paper proceeds as though no computational benefit is gained.

We can now present an algorithm that uses this idea of choosing one query to generalise at a time. First we introduce some variables:-

NEW The set of queries that have produced by a generalisation, and are to be processed using subsumption to find their extension.

SPARE The set of queries that produced null extensions, but have yet to be generalised.

CURRENT The next query to apply generalisation rules to.

Figure 7 shows the basic algorithm. The key point is that each time round, when new queries are produced by generalisation in step 5, only one query from the previous round is generalised to produce more queries that are tested to see if they yield an answer. If any do, then it is left to the individual application as to whether one or all non-null answers are presented. Note that as presented, the algorithm allows for a query to be selected for generalisation, even it was passed over at an earlier stage.

1. Initially we have no queries that have been tested for answers, so SPARE is empty. We only have one query, the original one, that needs to be tested for an answer.
2. Subject all queries in NEW to the process of subsumption to see if there is any non-null answer indicating that we can stop.
3. Since no queries in NEW produced answers, we will add them to SPARE so that they may possibly be selected later for generalisation.
4. Select a query from SPARE to be CURRENT
5. Produce NEW by applying all possible generalisations to CURRENT. Even if we are only going to select one query to be generalised from the new set, we still need all queries to see if an answer can be produced this round

It is possible to extend this algorithm. The first option is to allow interaction with the user. Instead of just giving the final answer, the user is allowed the option of rejecting it, whereupon the algorithm will discard NEW, and move to step 4. At this stage, CURRENT may be selected from any generalisation produced so far. It may be preferable to use some more limited form of backtracking, where SPARE is treated as a stack of sets, and step 3 pushes the set NEW onto SPARE, and step 4 is an incremental pop process, where CURRENT is chosen and removed from the set on the top of the stack, and once the top set becomes empty, it is implicitly popped.

7 Future work

This section is essentially a wish list of things that it would be good to work on, but there simply wasn't the time. There are many further directions that the work in this report can be extended. The first and most obvious is modifying the CLASSIC system to perform query generalisation. This would provide a platform with which to experiment with the ideas presented in this report.

The heuristic techniques described should be regarded as merely a starting point for a more developed system. As mentioned, the choice of preference ordering is highly subjective. If backtracking is used with this algorithm, then the user is providing feedback on his or her preferences

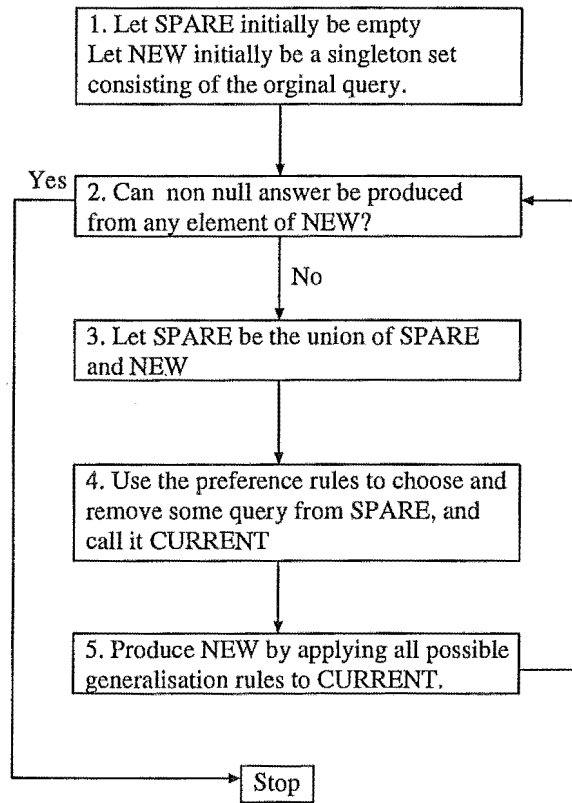


Figure 7: Algorithm to limit the complexity of generalisation by one query at a time to generalise.

regarding a good answer. At the very least, this can provide a way to reorder the generalisation preferences, but a more sophisticated approach would be to use higher-order forms where generalisation rankings are based on what generalisations have been performed previously.

Another direction to consider is choosing generalisations to make based not only the type of rule being applied, but on which attributes the rule is being applied to. Using Figure 5 as an example, relaxing the restrictions on advisor in preference to those on course load. Obviously, it is impossible to produce some universal ordering, as attributes vary between databases, but it should be possible to incorporate this idea into the concept of using user responses to build an ordering for an individual database.

This paper has presented only an informal discussion of query generalisation complexity, concentrating on upper bounds. A more general and rigorous analysis would be worthwhile.

This paper has only applied generalisations to concept expressions. In CLASSIC, queries can be made about individuals as well as concepts. Something to consider is the extension of generalisation to individual expressions, and perhaps even to the data manipulation language itself.

8 Conclusions

An underlying theme in this paper has been that any real world system will have the potential to be very large, and that complexity must be kept under control. The idea of imprecise querying has been applied to CLASSIC in order to provide a DBMS that can generalise queries, but still remain computationally feasible.

It has been pointed out the any imprecise query system is going to be potentially intractable unless some simplifying measures are taken. A collection of heuristics has been presented that achieves this. They are simple but can be improved by having modifications made on the fly based on user response. No consideration has been made regarding space complexity which may be a problem.

References

- [1] Tarek M. Anwar, Howard W. Beck, and Shamkant B. Navathe. Knowl-

- edge mining by imprecise querying: A classification-based approach. In *Proceedings of the IEEE Eighth International Conference on Data Engineering*, Tempe, Arizona, 1992. IEEE.
- [2] Howard W. Beck, Sunit K. Gala, and Shamkant B. Navathe. Classification as a query processing technique in the candidate semantic data model. In *Proceedings of the IEEE Fifth International Conference on Data Engineering*, Los Angeles, California, 1989.
 - [3] Alexander Borgida, Ronald J. Bachman, Deborah L. McGuinness, and Lori Alperin Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 58–67, Portland, Oregon, June 1989. ACM.
 - [4] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 34–37, Austin, Texas, August 1984. AAAI.
 - [5] T. Ichikawa and M. Hirikawa. ARES: a relational database with the capability of performing flexible interpretation of queries. *IEEE Transactions on Software Engineering*, SE-12(5):624–634, May 1986.
 - [6] H. J. Levesque and R. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3(2):78–93, May 1987.
 - [7] Ryszard S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, February 1983.
 - [8] A. Motro. SEAVE: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4(4), October 1986.
 - [9] A. Motro. VAGUE: a user interface to relational databases that permits vague queries. *ACM Transactions on Office Information Systems*, 6(3):187–214, July 1988.
 - [10] A. Motro. Accommodating imprecision in database systems. *SIGMOD RECORD*, 19(4), December 1990.

- [11] B. Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371–383, April 1988.
- [12] P.F. Patel-Schneider. *Decidable Logic-Based Knowledge representation*. PhD thesis, Department of Computer Science, University of Toronto, February 1987. A slightly revised and extended version is available as Technical Report Number 56, Schlumberger Palo Alto Research, May 1987 and as Technical Report 201/87, Department of Computer Science, University of Toronto.
- [13] E. Rosch. Principles of categorization. In E. Rosch and B. Lloyd, editors, *Cognition and Categorization*. Lawrence Erlbaum Associates, 1978.
- [14] William A. Woods and James G. Schmolze. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2-3):133–177, 1992.
- [15] M. Zemankova and A. Kandel. Implementing imprecision in information systems. *Information Sciences*, 37(1,2,3):107–141, December 1985.